What is Machine Learning?

What is Machine Learning?

Two definitions of Machine Learning are offered. Arthur Samuel described it as: "the field of study that gives computers the ability to learn without being explicitly programmed." This is an older, informal definition.

Tom Mitchell provides a more modern definition: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E."

Example: playing checkers.

E = the experience of playing many games of checkers

T = the task of playing checkers.

P = the probability that the program will win the next game.

In general, any machine learning problem can be assigned to one of two broad classifications:

Supervised learning and Unsupervised learning.

# Supervised Learning

## Supervised Learning

In supervised learning, we are given a data set **and already know what our correct output should look like**, having the idea that there is a relationship between the input and the output.

Supervised learning problems are categorized into **"regression" and "classification**" problems. In a regression problem, we are trying to predict results within **a continuous** output, meaning that we are trying to map input variables to some continuous function. In a classification problem, we are instead trying to predict results in **a discrete** output. In other words, we are trying to map input variables into discrete categories.

# Unsupervised Learning

## Unsupervised Learning

Unsupervised learning allows us to approach problems with little or no idea what our results should look like. We can derive structure from data where we don't necessarily know the effect of the variables.

We can derive this structure by clustering the data based on relationships among the variables in the data.

With unsupervised learning there is no feedback based on the prediction results.

**Example:**

Clustering: Take a collection of 1,000,000 different genes, and find a way to automatically group these genes into groups that are somehow similar or related by different variables, such as lifespan, location, roles, and so on.

Non-clustering: The "Cocktail Party Algorithm", allows you to find structure in a chaotic environment. (i.e. identifying individual voices and music from a mesh of sounds at a [cocktail party](https://en.wikipedia.org/wiki/Cocktail_party_effect)).

# Model Representation

To establish notation for future use, we’ll use x^{(i)}*x*(*i*) to denote the “input” variables (living area in this example), also called input features, and y^{(i)}*y*(*i*) to denote the “output” or target variable that we are trying to predict (price). A pair (x^{(i)} , y^{(i)} )(*x*(*i*),*y*(*i*)) is called a training example, and the dataset that we’ll be using to learn—a list of m training examples {(x^{(i)} , y^{(i)} ); i = 1, . . . , m}(*x*(*i*),*y*(*i*));*i*=1,...,*m*—is called a training set. Note that the superscript “(i)” in the notation is simply an index into the training set, and has nothing to do with exponentiation. We will also use X to denote the space of input values, and Y to denote the space of output values. In this example, X = Y = ℝ.

To describe the supervised learning problem slightly more formally, our goal is, given a training set, to learn a function h : X → Y so that h(x) is a “good” predictor for the corresponding value of y. For historical reasons, this function h is called a hypothesis. Seen pictorially, the process is therefore like this:



When the target variable that we’re trying to predict is continuous, such as in our housing example, we call the learning problem a regression problem. When y can take on only a small number of discrete values (such as if, given the living area, we wanted to predict if a dwelling is a house or an apartment, say), we call it a classification problem.

# Cost Function

We can measure the accuracy of our hypothesis function by using a **cost function**. This takes an average difference (actually a fancier version of an average) of all the results of the hypothesis with inputs from x's and the actual output y's.

J(\theta\_0, \theta\_1) = \dfrac {1}{2m} \displaystyle \sum \_{i=1}^m \left ( \hat{y}\_{i}- y\_{i} \right)^2 = \dfrac {1}{2m} \displaystyle \sum \_{i=1}^m \left (h\_\theta (x\_{i}) - y\_{i} \right)^2*J*(*θ*0​,*θ*1​)=2*m*1​*i*=1∑*m*​(*y*^​*i*​−*yi*​)2=2*m*1​*i*=1∑*m*​(*hθ*​(*xi*​)−*yi*​)2

# Gradient Descent

So we have our hypothesis function and we have a way of measuring how well it fits into the data. Now we need to estimate the parameters in the hypothesis function. That's where gradient descent comes in.

Imagine that we graph our hypothesis function based on its fields \theta\_0*θ*0​ and \theta\_1*θ*1​ (actually we are graphing the cost function as a function of the parameter estimates). We are not graphing x and y itself, but the parameter range of our hypothesis function and the cost resulting from selecting a particular set of parameters.

We put \theta\_0*θ*0​ on the x axis and \theta\_1*θ*1​ on the y axis, with the cost function on the vertical z axis. The points on our graph will be the result of the cost function using our hypothesis with those specific theta parameters. The graph below depicts such a setup.



We will know that we have succeeded when our cost function is at the very bottom of the pits in our graph, i.e. when its value is the minimum. The red arrows show the minimum points in the graph.

The way we do this is by taking the derivative (the tangential line to a function) of our cost function. The slope of the tangent is the derivative at that point and it will give us a direction to move towards. We make steps down the cost function in the direction with the steepest descent. The size of each step is determined by the parameter α, which is called the learning rate.

For example, the distance between each 'star' in the graph above represents a step determined by our parameter α. A smaller α would result in a smaller step and a larger α results in a larger step. The direction in which the step is taken is determined by the partial derivative of J(\theta\_0,\theta\_1)*J*(*θ*0​,*θ*1​). Depending on where one starts on the graph, one could end up at different points. The image above shows us two different starting points that end up in two different places.

The gradient descent algorithm is:

repeat until convergence:

\theta\_j := \theta\_j - \alpha \frac{\partial}{\partial \theta\_j} J(\theta\_0, \theta\_1)*θj*​:=*θj*​−*α*∂*θj*​∂​*J*(*θ*0​,*θ*1​)

where

j=0,1 represents the feature index number.

At each iteration j, one should simultaneously update the parameters \theta\_1, \theta\_2,...,\theta\_n*θ*1​,*θ*2​,...,*θn*​. Updating a specific parameter prior to calculating another one on the j^{(th)}*j*(*th*) iteration would yield to a wrong implementation.



# Gradient Descent Intuition

In this video we explored the scenario where we used one parameter \theta\_1*θ*1​ and plotted its cost function to implement a gradient descent. Our formula for a single parameter was :

Repeat until convergence:

|  |
| --- |
| \theta\_1:=\theta\_1-\alpha \frac{d}{d\theta\_1} J(\theta\_1)*θ*1​:=*θ*1​−*αdθ*1​*d*​*J*(*θ*1​) |

Regardless of the slope's sign for \frac{d}{d\theta\_1} J(\theta\_1)*dθ*1​*d*​*J*(*θ*1​), \theta\_1*θ*1​ eventually converges to its minimum value. The following graph shows that when the slope is negative, the value of \theta\_1*θ*1​ increases and when it is positive, the value of \theta\_1*θ*1​ decreases.



On a side note, we should adjust our parameter \alpha*α* to ensure that the gradient descent algorithm converges in a reasonable time. Failure to converge or too much time to obtain the minimum value imply that our step size is wrong.



### **How does gradient descent converge with a fixed step size \alpha*α*?**

The intuition behind the convergence is that \frac{d}{d\theta\_1} J(\theta\_1)*dθ*1​*d*​*J*(*θ*1​) approaches 0 as we approach the bottom of our convex function. At the minimum, the derivative will always be 0 and thus we get:

|  |
| --- |
| \theta\_1:=\theta\_1-\alpha \* 0*θ*1​:=*θ*1​−*α*∗0 |



# Classification

To attempt classification, one method is to use linear regression and map all predictions greater than 0.5 as a 1 and all less than 0.5 as a 0. However, this method doesn't work well because classification is not actually a linear function.

The classification problem is just like the regression problem, except that the values we now want to predict take on only a small number of discrete values. For now, we will focus on the **binary classification** **problem** in which y can take on only two values, 0 and 1. (Most of what we say here will also generalize to the multiple-class case.) For instance, if we are trying to build a spam classifier for email, then x^{(i)}*x*(*i*) may be some features of a piece of email, and y may be 1 if it is a piece of spam mail, and 0 otherwise. Hence, y∈{0,1}. 0 is also called the negative class, and 1 the positive class, and they are sometimes also denoted by the symbols “-” and “+.” Given x^{(i)}*x*(*i*), the corresponding y^{(i)}*y*(*i*) is also called the label for the training example.

# Cost Function

We cannot use the same cost function that we use for linear regression because the Logistic Function will cause the output to be wavy, causing many local optima. In other words, it will not be a convex function.

Instead, our cost function for logistic regression looks like:

|  |
| --- |
| *J*(*θ*)=1*m*∑*i*=1*m*Cost(*hθ*(*x*(*i*)),*y*(*i*))Cost(*hθ*(*x*),*y*)=−log(*hθ*(*x*))Cost(*hθ*(*x*),*y*)=−log(1−*hθ*(*x*))if y = 1if y = 0 |

When y = 1, we get the following plot for J(\theta)*J*(*θ*) vs h\_\theta (x)*hθ*​(*x*):



Similarly, when y = 0, we get the following plot for J(\theta)*J*(*θ*) vs h\_\theta (x)*hθ*​(*x*):



|  |
| --- |
| Cost(*hθ*(*x*),*y*)=0 if *hθ*(*x*)=*y*Cost(*hθ*(*x*),*y*)→∞ if *y*=0and*hθ*(*x*)→1Cost(*hθ*(*x*),*y*)→∞ if *y*=1and*hθ*(*x*)→0 |

If our correct answer 'y' is 0, then the cost function will be 0 if our hypothesis function also outputs 0. If our hypothesis approaches 1, then the cost function will approach infinity.

If our correct answer 'y' is 1, then the cost function will be 0 if our hypothesis function outputs 1. If our hypothesis approaches 0, then the cost function will approach infinity.

Note that writing the cost function in this way guarantees that J(θ) is convex for logistic regression.

# Simplified Cost Function and Gradient Descent

**Note:** [6:53 - the gradient descent equation should have a 1/m factor]

We can compress our cost function's two conditional cases into one case:

\mathrm{Cost}(h\_\theta(x),y) = - y \; \log(h\_\theta(x)) - (1 - y) \log(1 - h\_\theta(x))Cost(*hθ*​(*x*),*y*)=−*y*log(*hθ*​(*x*))−(1−*y*)log(1−*hθ*​(*x*))

Notice that when y is equal to 1, then the second term (1-y)\log(1-h\_\theta(x))(1−*y*)log(1−*hθ*​(*x*)) will be zero and will not affect the result. If y is equal to 0, then the first term -y \log(h\_\theta(x))−*y*log(*hθ*​(*x*)) will be zero and will not affect the result.

We can fully write out our entire cost function as follows:

|  |
| --- |
| J(\theta) = - \frac{1}{m} \displaystyle \sum\_{i=1}^m [y^{(i)}\log (h\_\theta (x^{(i)})) + (1 - y^{(i)})\log (1 - h\_\theta(x^{(i)}))]*J*(*θ*)=−*m*1​*i*=1∑*m*​[*y*(*i*)log(*hθ*​(*x*(*i*)))+(1−*y*(*i*))log(1−*hθ*​(*x*(*i*)))] |

A vectorized implementation is:

|  |
| --- |
| *h*=*g*(*Xθ*)*J*(*θ*)=1*m*⋅(−*yT*log(*h*)−(1−*y*)*T*log(1−*h*)) |

### **Gradient Descent**

Remember that the general form of gradient descent is:

|  |
| --- |
| *Repeat*{*θj*:=*θj*−*α*∂∂*θjJ*(*θ*)} |

We can work out the derivative part using calculus to get:

|  |
| --- |
| *Repeat*{*θj*:=*θj*−*αm*∑*i*=1*m*(*hθ*(*x*(*i*))−*y*(*i*))*x*(*i*)*j*} |

Notice that this algorithm is identical to the one we used in linear regression. We still have to simultaneously update all values in theta.

A vectorized implementation is:

\theta := \theta - \frac{\alpha}{m} X^{T} (g(X \theta ) - \vec{y})*θ*:=*θ*−*mα*​*XT*(*g*(*Xθ*)−*y*​)

# Multiclass Classification: One-vs-all

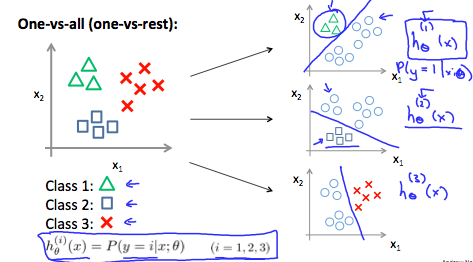
Now we will approach the classification of data when we have more than two categories. Instead of y = {0,1} we will expand our definition so that y = {0,1...n}.

Since y = {0,1...n}, we divide our problem into n+1 (+1 because the index starts at 0) binary classification problems; in each one, we predict the probability that 'y' is a member of one of our classes.

|  |
| --- |
| *y*∈{0,1...*n*}*h*(0)*θ*(*x*)=*P*(*y*=0|*x*;*θ*)*h*(1)*θ*(*x*)=*P*(*y*=1|*x*;*θ*)⋯*h*(*n*)*θ*(*x*)=*P*(*y*=*n*|*x*;*θ*)prediction=max*i*(*h*(*i*)*θ*(*x*)) |

We are basically choosing one class and then lumping all the others into a single second class. We do this repeatedly, applying binary logistic regression to each case, and then use the hypothesis that returned the highest value as our prediction.

The following image shows how one could classify 3 classes:



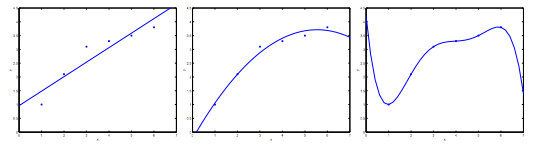
**To summarize:**

Train a logistic regression classifier h\_\theta(x)*hθ*​(*x*) for each class￼ to predict the probability that ￼ ￼y = i￼ ￼.

To make a prediction on a new x, pick the class ￼that maximizes h\_\theta (x) *hθ*​(*x*)

The Problem of Overfitting

Consider the problem of predicting y from x ∈ R. The leftmost figure below shows the result of fitting a y = θ\_0 + θ\_1x*θ*0​+*θ*1​*x* to a dataset. We see that the data doesn’t really lie on straight line, and so the fit is not very good.



Instead, if we had added an extra feature x^2*x*2 , and fit y = \theta\_0 + \theta\_1x + \theta\_2x^2*y*=*θ*0​+*θ*1​*x*+*θ*2​*x*2 , then we obtain a slightly better fit to the data (See middle figure). Naively, it might seem that the more features we add, the better. However, there is also a danger in adding too many features: The rightmost figure is the result of fitting a 5^{th}5*th* order polynomial y = \sum\_{j=0} ^5 \theta\_j x^j*y*=∑*j*=05​*θj*​*xj*. We see that even though the fitted curve passes through the data perfectly, we would not expect this to be a very good predictor of, say, housing prices (y) for different living areas (x). Without formally defining what these terms mean, we’ll say the figure on the left shows an instance of **underfitting**—in which the data clearly shows structure not captured by the model—and the figure on the right is an example of **overfitting**.

Underfitting, or high bias, is when the form of our hypothesis function h maps poorly to the trend of the data. It is usually caused by a function that is too simple or uses too few features. At the other extreme, overfitting, or high variance, is caused by a hypothesis function that fits the available data but does not generalize well to predict new data. It is usually caused by a complicated function that creates a lot of unnecessary curves and angles unrelated to the data.

This terminology is applied to both linear and logistic regression. There are two main options to address the issue of overfitting:

1) Reduce the number of features:

* Manually select which features to keep.
* Use a model selection algorithm (studied later in the course).

2) Regularization

* Keep all the features, but reduce the magnitude of parameters \theta\_j*θj*​.
* Regularization works well when we have a lot of slightly useful features.

# Cost Function

**Note:** [5:18 - There is a typo. It should be \sum\_{j=1}^{n} \theta \_j ^2∑*j*=1*n*​*θj*2​ instead of \sum\_{i=1}^{n} \theta \_j ^2∑*i*=1*n*​*θj*2​]

If we have overfitting from our hypothesis function, we can reduce the weight that some of the terms in our function carry by increasing their cost.

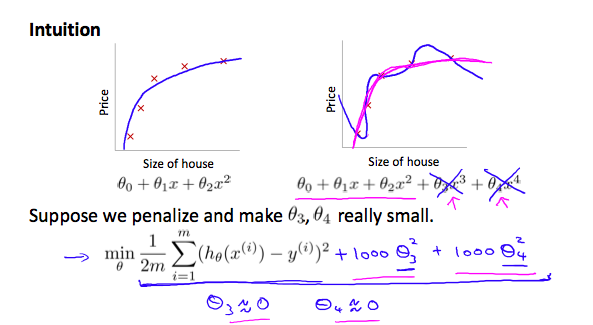
Say we wanted to make the following function more quadratic:

\theta\_0 + \theta\_1x + \theta\_2x^2 + \theta\_3x^3 + \theta\_4x^4*θ*0​+*θ*1​*x*+*θ*2​*x*2+*θ*3​*x*3+*θ*4​*x*4

We'll want to eliminate the influence of \theta\_3x^3*θ*3​*x*3 and \theta\_4x^4*θ*4​*x*4 . Without actually getting rid of these features or changing the form of our hypothesis, we can instead modify our **cost function**:

min\_\theta\ \dfrac{1}{2m}\sum\_{i=1}^m (h\_\theta(x^{(i)}) - y^{(i)})^2 + 1000\cdot\theta\_3^2 + 1000\cdot\theta\_4^2*minθ*​ 2*m*1​∑*i*=1*m*​(*hθ*​(*x*(*i*))−*y*(*i*))2+1000⋅*θ*32​+1000⋅*θ*42​

We've added two extra terms at the end to inflate the cost of \theta\_3*θ*3​ and \theta\_4*θ*4​. Now, in order for the cost function to get close to zero, we will have to reduce the values of \theta\_3*θ*3​ and \theta\_4*θ*4​ to near zero. This will in turn greatly reduce the values of \theta\_3x^3*θ*3​*x*3 and \theta\_4x^4*θ*4​*x*4 in our hypothesis function. As a result, we see that the new hypothesis (depicted by the pink curve) looks like a quadratic function but fits the data better due to the extra small terms \theta\_3x^3*θ*3​*x*3 and \theta\_4x^4*θ*4​*x*4.



We could also regularize all of our theta parameters in a single summation as:

|  |
| --- |
| min\_\theta\ \dfrac{1}{2m}\ \sum\_{i=1}^m (h\_\theta(x^{(i)}) - y^{(i)})^2 + \lambda\ \sum\_{j=1}^n \theta\_j^2*minθ*​ 2*m*1​ ∑*i*=1*m*​(*hθ*​(*x*(*i*))−*y*(*i*))2+*λ* ∑*j*=1*n*​*θj*2​ |

The λ, or lambda, is the **regularization parameter**. It determines how much the costs of our theta parameters are inflated.

Using the above cost function with the extra summation, we can smooth the output of our hypothesis function to reduce overfitting. If lambda is chosen to be too large, it may smooth out the function too much and cause underfitting. Hence, what would happen if \lambda = 0*λ*=0 or is too small ?

# Regularized Linear Regression

**Note:** [8:43 - It is said that X is non-invertible if m \leq≤ n. The correct statement should be that X is non-invertible if m < n, and may be non-invertible if m = n.

We can apply regularization to both linear regression and logistic regression. We will approach linear regression first.

Regulation is to prevent Theta values too big thus cause over fitting

### **Gradient Descent**

We will modify our gradient descent function to separate out \theta\_0*θ*0​ from the rest of the parameters because we do not want to penalize \theta\_0*θ*0​.

|  |
| --- |
| Repeat {    *θ*0:=*θ*0−*α* 1*m* ∑*i*=1*m*(*hθ*(*x*(*i*))−*y*(*i*))*x*(*i*)0    *θj*:=*θj*−*α* [(1*m* ∑*i*=1*m*(*hθ*(*x*(*i*))−*y*(*i*))*x*(*i*)*j*)+*λmθj*]}          *j*∈{1,2...*n*} |

The term \frac{\lambda}{m}\theta\_j*mλ*​*θj*​ performs our regularization. With some manipulation our update rule can also be represented as:

\theta\_j := \theta\_j(1 - \alpha\frac{\lambda}{m}) - \alpha\frac{1}{m}\sum\_{i=1}^m(h\_\theta(x^{(i)}) - y^{(i)})x\_j^{(i)}*θj*​:=*θj*​(1−*αmλ*​)−*αm*1​∑*i*=1*m*​(*hθ*​(*x*(*i*))−*y*(*i*))*xj*(*i*)​

The first term in the above equation, 1 - \alpha\frac{\lambda}{m}1−*αmλ*​ will always be less than 1. Intuitively you can see it as reducing the value of \theta\_j*θj*​ by some amount on every update. Notice that the second term is now exactly the same as it was before.

### **Normal Equation**

Now let's approach regularization using the alternate method of the non-iterative normal equation.

To add in regularization, the equation is the same as our original, except that we add another term inside the parentheses:

|  |
| --- |
| *θ*=(*XTX*+*λ*⋅*L*)−1*XTy*where  *L*=⎡⎣⎢⎢⎢⎢⎢⎢011⋱1⎤⎦⎥⎥⎥⎥⎥⎥ |

L is a matrix with 0 at the top left and 1's down the diagonal, with 0's everywhere else. It should have dimension (n+1)×(n+1). Intuitively, this is the identity matrix (though we are not including x\_0*x*0​), multiplied with a single real number λ.

Recall that if m < n, then X^TX*XTX* is non-invertible. However, when we add the term λ⋅L, then X^TX*XTX* + λ⋅L becomes invertible

